Refine Search

Search Results -

Terms	Documents
L9 and wholesale	9

Database: US Pre-Grant Publication Full-Text Database; US Patents Full-Text Database; US OCR Full-Text Database; EPO Abstracts Database; JPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Search:

Refine Search | Recall Text | Clear | Interrupt

Search History

DATE: Saturday, March 04, 2006 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR		
L10	L9 and wholesale	9	L10
L9	L8 and trad$	76	L9
L8	L7 and (loan with pool or loan near pool or loan adj pool)	84	L8
L7	loans and underwriting	616	L7
L6	L5 and automobile near loans	4	L6
L5	L4 and personal near loans	34	L5
L4	credit near lines	2568	L4
	DB=USPT; PLUR=YES; OP=OR		
L3	(5809483 | 5794207 | 5809478 | 5794219 | 5375055 | 5611052 | 5893079 | 6026364 | 4903201 | 5835896 | 5884286 | 5940812 | 5704045 | 5774883 | 6119093 | 5895454 | 5742775 | 5873071 | 5890138 | 5193056 | 5845265 | 5845266)![PN]	22	L3
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR		

L2	('6594635')[ABPN1,NRPN,PN,TBAN,WKU]	2	L2
L1	6594635.pn.	2	L1

END OF SEARCH HISTORY

L7: Entry 34 of 45 File: USPT Oct 20, 1998

DOCUMENT-IDENTIFIER: US 5826076 A
**** See image for Certificate of Correction ****
TITLE: Computer-based information access method and apparatus to permit SQL-based manipulation of programming language-specific data files

Abstract Text (1):
A method and apparatus that permits creation, reading and modification of 3GL application programs by SQL requests. A catalog is created by selecting source programs from 3GL application data for which file definitions are to be extracted, identifying specific files within the selected source programs to be processed, extracting appropriate schema from the selected files, and recording the appropriate schema in a catalog. Once the 3GL-specific data schema contained in the application source has been extracted and stored in the catalog, the relational database which is to be based upon the data represented by the 3GL schema is defined and stored in the system catalog so that tables in the relational database may be accessed by a runtime SQL database engine. Further, both 3GL data and relational database data may be modified and maintained with a single tool set. After the catalog is created, the invention uses the catalog to process SQL requests in order to access the relational database representation of the 3GL data by parsing the SQL query request, generating a set of possible execution plans for manipulating the relational form of the data, selecting an optimal plan based upon information provided from the system catalog, and executing the plan by servicing the SQL request for relational data from the underlying 3GL data files described in the system catalog.

Brief Summary Text (3):
This invention relates to a method and apparatus for storing, accessing and manipulating information used in the automation of business data processing functions. Specifically, the invention addresses the problem of access via client computer-based application components using the SQL relational database query language to non-relational data files produced and maintained by server-based procedural applications written in a third-generation, general purpose programming language.

Brief Summary Text (7):
These efforts focused on the notion that the data files maintained by applications actually represented a "database" of business information. If a model could be found that would serve to raise the level of abstraction of this model high enough, it would be practical to access data in the database with much less specific programming skills. Many possible directions were pursued, but the most practical were what were called the network model, the hierarchical model and the relational model. The network and hierarchical data models led to some products designed to implement those database architectures. It is the relational model, however, that is of much greater interest, for it survived the test of time.

Brief Summary Text (8):
Perhaps the greatest attraction of relational database technology was that it was based on a mathematically rigorous theory that a database could be represented by a collection of relations (commonly called "tables") defined in such a way that they could be combined and operated on in certain well-defined ways giving a result that

was itself a relation. This property of closure permitted operations within this system to be of arbitrary complexity, yielding an extremely powerful means of manipulating data. Most importantly, at the time of database design there was much less need to anticipate the precise ways that data would be accessed and manipulated by the ultimate user of the information. Commercially viable database systems built on relational principles began to appear on the market in the early 1980's. One of the first commercial relational databases was one called "DB2" produced by IBM. It was based on an IBM research effort begun in the 1970's called "System R" that defined an English-like computer language, called SEQUEL, with which data extraction ("query") and manipulation ("update") operations on a relational database could be expressed. Within the DB2 product, this language was called SQL, which was an acronym that stood for Structured Query Language.

Brief Summary Text (9):
During the early days of DB2 and other Relational Database Management Systems (RDBMS) there were many problems achieving commercially practical results due to the complexity of efficiently processing the SQL operations. Eventually these problems were overcome for many classes of problems, and relational databases came into widespread use. During the early period, many variants of IBM's SQL were implemented by the companies providing RDBMSs. Recognizing the need to provide a standard variant of the language as being within its mission, the American National Standards Institute (ANSI) launched an effort in 1982 to produce a standard relational query language. This effort resulted in ANSI standard X3.135-1986, which was informally called SQL/86. Like most first attempts to standardize a computer language, SQL/86 was a "lowest common denominator" solution. It lacked many of the basic features present in most commercial RDBMSs, and there were many pre-existing systems available, so it was neither widely used as a model for commercial systems nor was it used as a target language by database application developers. Subsequently, to cure many of the deficiencies of SQL/86, ANSI developed a revised version which became known as SQL/89. Both versions of the SQL standard were adopted by the International Standards Organization (ISO) as well.

Brief Summary Text (10):
By 1990, the ANSI and ISO standards notwithstanding, dozens of variations of the SQL language were in common commercial use, and the economic advantage of achieving a broad level of real standardization became obvious. Yet the existing standards bodies (ANSI and ISO) were in the process of producing the third generation of the standard SQL, later to be called SQL/92. This effort reflected a vast expansion of the functions and language defined by previous versions of the standard, and it allowed great flexibility in implementation. Recognizing that very specific guidance was needed if the SQL standards were ever to serve as a point of convergence for the numerous commercial implementations, a group of database vendors formed the SQL Access Group, or SAG, to define target relational database functions, language and features that would enable interoperability of the databases and tools.

Brief Summary Text (11):
The first specification to come out of SAG was a language based on the SQL/89 language known as SAG SQL. Subsequent work by SAG has produced a "call-level interface", or CLI, for this language known as the SAG CLI. This call-level interface defines a standard way for general-purpose programming languages to access the database via the capabilities of dynamically generated SQL. The SAG CLI formed the basis for a specification by Microsoft Corporation, the owner of software products used most commonly for desktop operating environments, DOS and Windows. This specification was called Open DataBase Connectivity, or ODBC. Almost immediately, another group of vendors led by Borland International, defined a similar common interface for desktop connectivity of application tools to enterprise data. Both of the specifications used SQL as the basis for data access. Virtually every major front end client tool vendor pledged support to one or both of these specifications. As of early 1994, most key vendors had released products

based on the ODBC specification. The industry's dream of a single connection point to business information appeared to have been realized.

Brief Summary Text (12):
Even today, however, in spite of the widespread acceptance of relational database technology, the majority of information used by existing business applications is stored by and accessed through the use of older procedural computer programs written in 3GLs such as COBOL and PL/I. This means that, without enabling technology, the majority of businesses cannot fully exploit the opportunity presented by universal desktop client connectivity via SQL-based standards, since most of the data was not under the control of a relational database.

Brief Summary Text (14):
The present invention solves the problem of providing SQL-based client application access to data files created, read and modified by 3GL application programs. It does so by providing a method and apparatus that allows the data files to be described as a set of relational tables, the resulting description to be stored in a system catalog, and the data files to be manipulated by SQL operations as relational tables in exactly the same manner as by 3GL source-based applications. The SQL-based data file manipulation capability provided by the present invention is capable of supporting any of a class of 3GL (such as COBOL) data files and record structures present in existing applications. In the case of COBOL, this includes support for all COBOL file organizations (i.e., sequential, relative and indexed), fixed and variable length records, COBOL file record numbers and keys, multiple COBOL record types in a single file, redefinition of portions of the data record (such redefinitions being possibly nested within other redefinitions), repeating items (both group and elementary) and all of the COBOL elementary data types (e.g., binary, packed decimal, display text, display numeric, edited text and edited numeric). The data are operated on in precisely the same way as the normal COBOL file and arithmetic subsystems would operate on them.

Brief Summary Text (15):
The present invention provides for means to recover 3GL non-relational data schema information from the source code of existing applications, and to store this schema information in a disk-based catalog structure outside of the original source code. Once the 3GL-specific data schema contained in the application source has been extracted and stored in the catalog, the relational database to be based upon the data represented by the 3GL schema is defined and stored in the system catalog such that tables in the relational database can be accessed by a runtime SQL database engine. Furthermore, the present invention permits both the 3GL data schema and the relational database schema to be modified and maintained within a single tool set. Subsequent to any modification of the 3GL schema represented in the catalog by the tool set, the present invention provides the means to "export" the modified schema information in the source form of the original 3GL programming language.

Brief Summary Text (16):
The present invention provides for means, at the time of client application execution, to process SQL requests in order to access the relational database representation of the 3GL application data. This is accomplished by parsing the SQL-form of the query request, generating a set of possible execution plans for manipulating the relational form of the data, selecting an optimal plan based on information provided from the system catalog, and executing the plan by servicing the requests for relational data from the underlying 3GL data files described in the system catalog. In the case of SQL operations that modify the data contained within the relational database, the present invention provides for means to alter the data within the 3GL data files.

Brief Summary Text (17):
In particular, the method of the present invention for processing SQL requests to access a relational database representation of 3GL application data, comprises,

parsing the SQL requests, generating a plurality of execution plans for manipulating the relational database representation of the 3GL application data, selecting an execution plan as a function of information provided in a predetermined catalog which relates SQL requests to 3GL application data, and finally, executing the selected execution plan by servicing the SQL request to access the relational database representation of the 3GL application data, as a function of the information in the predetermined catalog.

Brief Summary Text (19):
Yet further, the executing step of the invention may comprise, scanning all rows of the predetermined catalog to extract filter predicates, and sorting the rows of the catalog in accordance with the extracted filter predicates to form a query to access the relational database representation of the 3GL application data.

Detailed Description Text (2):
In the exemplary embodiment of the present invention, data contained within files maintained by application programs written in the COBOL programming language are made accessible to client applications using the SQL-based application programming interface known as ODBC, or Open DataBase Connectivity, used within the Microsoft Windows operating system. This configuration of SQL and 3GL-specific data is typical of a large class of data access problems to which the present invention is applicable.

Detailed Description Text (4):
Referring to FIG. 2, a block diagram illustrating the key components and data flow between the components of the present invention, block 201, and surrounding data processing elements, blocks 100, 101 and 206, is shown. Prior to the runtime SQL manipulation of the COBOL data, the database schema information necessary to enable this process must be recorded in a the system catalog, shown in block 205. The definition and maintenance of this schema information is controlled by an interactive application program, or system catalog maintenance tool, represented by block 203 of FIG. 2. The original COBOL application program source is shown in block 206. Within this source code, the data file definition information is extracted under the control of the system catalog maintenance tool by the schema recovery tool illustrated in block 204. Within the program processes performed in block 204, the source programs from which to extract file definitions are selected, the specific files to be processed are identified and the appropriate schema information recorded in the system catalog, block 205. The relational database schema for the COBOL database management system is then described by the COBOL application developer with the facilities of the system catalog maintenance tool. After the system catalog information for the COBOL database is complete, SQL requests to manipulate the original data shown in block 101 may be processed and the appropriate operations and transformations performed on the data files. This processing of SQL and the resulting operations on the data files are performed within the program logic represented by block 202, the COBOL RDBMS engine, shown in detail in FIG. 11

Detailed Description Text (5):
Turning now to the expanded block diagram of the system catalog component of the exemplary embodiment of the present invention (block 205 FIG. 2), FIG. 3 shows the general structure and type of information recorded. The system catalog contains all of the COBOL file schema information contained in the application source program, shown in block 301. This includes information found in the COBOL environment division SELECT statement, such as primary and alternate key definitions (block 307) and file name and organization (block 302). Also included in this part of the system catalog is all of the file information found in the COBOL data division file section. The logical file and record definitions (block 302) and the nested definitions of data items within records (block 303) are contained in this part of the system catalog. This includes group (block 304) and elementary level (block 305) data items and any condition items (block 306) defined on the data items. Also

contained in the system catalog is information necessary to define the relational database schema representation of the COBOL file set (block 301) defined in block 309. As such, this relational schema includes information to define the tables within the database (block 310). The definition of each table requires information to be included in the system catalog to define the columns comprising each table (block 311), the indices defined on each table (block 312) and information that is important to the proper operation of the present invention, "predicates" (block 313). Each column of each table in the database definition is "bound" to a source of its value defined within the COBOL file definition portion of the system catalog. For most columns, the binding is to a particular data item, but for some it is to an "ordinal" value that represents an occurrence number of a repeating COBOL item. As COBOL data records are read by the COBOL relation manager described later, each record is tested against a set of table "predicates" that make up a filter for the table. This facility permits the COBOL redefinition of data areas to be supported by the present invention. Information regarding these predicates is contained in the system catalog shown.

Detailed Description Text (8):
In block 405 the catalog maintenance tool (block 203 FIG. 2) presents the list of programs available in the connected object file to the user. The user selects zero or more program names to be processed. The selected list is then used in the loop that begins with block 406. If there are one or more programs to process, then as indicated in block 407, the next program to process is selected and processing occurs as shown in FIG. 5. Upon return, the processed program is removed from the list and the process loops back to block 406. When the list of programs is empty, the catalog maintenance tool disconnects from the COBOL application program file in block 408, thereby allowing for closing the COBOL object file and the release of data structures allocated for processing programs contained in that object file, thus ending the connection established in block 403. Then the COBOL schema recovery process completes at block 409 by returning control to the catalog maintenance tool. At this point the catalog maintenance allows the user to begin another import, to define relational tables on the COBOL schema as shown in block 303 of FIG. 3, to do assorted other tasks related to the catalog or data described by the catalog, or to terminate the process.

Detailed Description Text (10):
FIG. 6 is the third of three flow charts showing how the COBOL schema is recovered for use in the catalog shown in FIG. 3 block 301. When a particular program within an object file has been selected (see FIG. 4) and a particular file within that program has been selected for addition to a particular catalog (see FIG. 5), control reaches block 601. The specified catalog file name is used to connect to the catalog as shown in block 602. The catalog file is opened and validated to be a catalog. If the catalog file does not exist, this is indicated to the catalog maintenance tool (block 203 FIG. 2), which gives the user an opportunity to specify a different catalog file name or to request creation of the catalog file. If the user requests creation of a catalog file, then a new catalog file is created, initialized, and connected as the currently connected catalog. Once a catalog is successfully connected, the specified COBOL file-name is compared against the names of files described in the connected catalog as indicated in block 603. If the COBOL file-name is already in the catalog, this is indicated to the catalog maintenance tool, which provides the user an opportunity to skip the file or specify an alias name for the file to be used in the catalog as shown in block 604. This guarantees that files described in the catalog have unique names for later use. For a selected COBOL file-name (or alias) that is not in the catalog, control reaches block 605. The file definition from the connected program for the specified COBOL file-name is added to the catalog, including descriptions of keys and key parts. If the file definition in the connected object program contains a literal file access name, then a file instance record is also added to the catalog; this file instance record specifies the literal file access name as the path string for a file instance. If the file description requires alphabet definitions, for code-set specification or

key collating sequence specification, then these alphabet definitions are also added to the catalog. In block 606, the record descriptions associated with the file are added to the catalog; this includes item definition records for each group, elementary, and condition item associated with the file. In block 607, repeating item descriptions are added to the catalog; this includes descriptions of each item associated with the file that is described with an OCCURS clause, including any ordering key items declared in those OCCURS clauses. This completes adding the file to the catalog. In block 608, the catalog maintenance tool disconnects from the catalog so that the user has the option of changing the catalog for the next file to be added to the catalog. Then control passes to block 609 where the process returns to the outer level (see FIG. 5, block 507).

Detailed Description Text (12):
FIG. 7 details the steps involved in the process of the COBOL developer's defining the tables that comprise a COBOL database to be accessed by the COBOL RDBMS engine component (block 202, FIG. 2) of the present invention. While many methods of interacting with the COBOL developer to accomplish this task are possible, the exemplary embodiment of the present invention uses a graphical user interface driven approach. Since the flow of control of such an interface is determined by the order in which the user decides to perform the steps of the process, FIGS. 7-10 reflect this non-deterministic control flow. In block 701 of FIG. 7, the user selects the next (or first) function he wishes to perform regarding the presently selected table within the presently selected database. The primary choices at this point are to transfer control to block 702 in order to add, edit or remove filter predicates defined for the table (detailed in FIG. 8); transfer control to block 703 in order to add a column defined for the table (detailed in FIG. 9), or to transfer control to block 704 in order to add or remove an index for the table (detailed in FIG. 10). When control returns from the selected function, the decision is made in block 705 as to whether or not the design for the selected table is complete at this time. If not, control returns to block 701. If so, control passes to block 706. Within block 706, the steps are performed to update the disk-based copy of the system catalog (block 205, FIG. 2) if necessary. The procedure then exits normally.

Detailed Description Text (16):
Turning now to the detailed charts of the COBOL RDBMS engine component of the exemplary embodiment of the present invention (block 202 FIG. 2), FIG. 11 shows the high level structure and subassemblies and FIGS. 12-13 illustrate the overall flow of control within this component of the exemplary embodiment of the present invention. As shown in FIG. 11, the client application (block 100, FIGS. 1 and 11) interacts with the COBOL RDBMS engine (block 202, FIGS. 2 and 11) primarily via SQL requests. The initial processing is performed by an ODBC SQL query engine (block 1101) which analyzes the request and determines an optimized, procedural plan of access to perform the query or update operation. This plan is then performed by the engine by requesting that operations on the tables within the COBOL relational database be performed by the COBOL relation manager, block 1102. The relation manager, in turn and with the information held in the memory resident image of the system catalog (block 205, FIGS. 2 and 11) utilizes the COBOL record manager, block 1103, in order to read, write, update and delete physical disk storage records in the COBOL data files (block 101, FIGS. 1 and 11) affected by the plan.

Detailed Description Text (28):
FIG. 20 details the steps performed to open the next table (see FIG. 14, block 1401). In predefined procedure block 2001, session and database IDs are mapped to a database. If the IDs are valid, decision block 2002 passes control to decision block 2003; otherwise, the procedure exits with an error. Block 2003 searches for a table definition with a matching name. If the table is found, decision block 2004 passes control to decision block 2005; otherwise, the procedure exits with an error. Decision block 2005 determines whether the table is a catalog table and if so, whether catalog tables are hidden, in which case the procedure exits with an

error. If the table is not a catalog table or if catalog tables are not hidden, control passes to decision block 2006. Decision block 2006 determines whether the table is a virtual table, in which case the procedure exits with an error. If the table is not a virtual table, control passes to decision block 2007. Decision block 2007 determines whether the table is a catalog table.

Detailed Description Text (38):
If decision block 2306 determines that the table is not a virtual table, control advances to decision block 2309, which determines whether the file is open. If the file is not open, control advances to block 2315. If the file is open, control passes to I/O block 2310, which closes the physical file. Block 2311 then decrements the open count in the file instance structure and passes control to decision block 2312. Decision block 2312 determines whether the open count equals zero. If not, control advances to block 2315. If the open count does equal zero, then control passes to block 2313, which delinks the file instance structure from the database structure. Predefined procedure block 2314 then deallocates the file instance structure and passes control to block 2315.

Detailed Description Text (46):
If decision block 2905 determines that this is the first key segment, decision block 2905 passes control to block 2906. Block 2906 sets the current key part number to zero and control advances to block 2908. Block 2908 fetches addresses of the column and item structures for the current key part and passes control to decision block 2909. Decision block 2909 determines whether the key part value is too large or too small. If the key part value is not an acceptable size, the procedure exits with an error. If the key part value is an acceptable size, control passes to block 2910, which copies the key part value to the key buffer (described in detail in FIG. 29B). Block 2911 then increments the key part number and the procedure exits normally.

Detailed Description Text (48):
FIG. 29B details the steps performed to copy the key part value to the key buffer (see FIG. 29A, block 2910). Block 2912 computes the address of the key part in the key buffer and passes control to decision block 2913. Decision block 2913 determines whether the column type equals text.

Detailed Description Text (49):
If decision block 2913 determines that the column type equals text, control passes to decision block 2914. Decision block 2914 determines whether the column is an ordinal column. If the column is an ordinal column, the procedure exits with an error. If the column is not an ordinal column, control passes to block 2915, which copies the key part value to the key buffer. Block 2916 then copies spaces to the key buffer in order to pad the value to the size of the key part and the procedure exits normally.

Detailed Description Text (50):
If decision block 2913 determines that the column type does not equal text, control advances to decision block 2917. Decision block 2917 determines whether the column type equals binary, vartext, or varbinary. If the column type equals binary, vartext, or varbinary, control passes to decision block 2918. Decision block 2918 determines whether the column is an ordinal column. If the column is an ordinal column, the procedure exits with an error. If the column is not an ordinal column, control passes to block 2919, which copies the key part value to the key buffer. Block 2920 then copies binary zeroes to the key buffer in order to pad the value to the size of the key part and the procedure exits normally.

Detailed Description Text (51):
If decision block 2917 determines that the column type does not equal binary, vartext, or varbinary, control advances to decision block 2921. Decision block 2921 determines whether the column type equals bit. If the column type equals bit,

control passes to block 2922, which sets binValue to equal the first byte of the key part value ANDed with 0x01. Block 2923 then puts binValue in arithmetic register #0 (described in detail in FIG. 38). Block 2924 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (52):
If decision block 2921 determines that the column type does not equal bit, control advances to decision block 2925. Decision block 2925 determines whether the column type equals unsigned byte. If the column type equals unsigned byte, control passes to block 2926, which sets binValue to equal the first byte of the key part value, zero-filled to 32 bits. Block 2927 then puts binValue in arithmetic register #0 (described in detail in FIG. 38). Block 2928 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (53):
If decision block 2925 determines that the column type does not equal unsigned byte, control passes to decision block 2929. Decision block 2929 determines whether the column type equals short. If the column type equals short, control passes to block 2930, which sets binValue to equal the first two bytes of the key part value, extended to 32 bits. Block 2931 then puts binValue in arithmetic register #0 (described in detail in FIG. 38). Block 2932 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (54):
If decision block 2929 determines that the column type does not equal short, control advances to decision block 2933. Decision block 2933 determines whether the column type equals long. If the column type equals long, control passes to block 2934, which puts the 32-bit key part value in arithmetic register #0 (described in detail in FIG. 38). Block 2935 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (55):
If decision block 2933 determines that the column type does not equal long, control passes to decision block 2936. Decision block 2936 determines whether the column type equals currency. If the column type equals currency, control passes to block 2937, which puts the currency-type key part value in arithmetic register #0 (described in detail in FIG. 39). Block 2938 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (56):
If decision block 2936 determines that the column type does not equal currency, control passes to decision block 2939. Decision block 2939 determines whether the column type equals date, time, or datetime. If the column type equals date, time, or datetime, control passes to block 2940, which puts the IEEE double precision key part value in arithmetic register #0 (described in detail in FIG. 40). Block 2941 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (57):
If decision block 2939 determines that the column type does not equal date, time, or datetime, control advances to decision block 2942. Decision block 2942 determines whether the column type equals double. If the column type equals double, control passes to block 2943, which puts the IEEE double precision key part value in arithmetic register #0 (described in detail in FIG. 40). Block 2944 then rounds the arithmetic register #0 to the size of the key buffer (described in detail in

FIG. 41). Block 2945 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (58):
If decision block 2942 determines that the column type does not equal double, control advances to decision block 2946. Decision block 2946 determines whether the column type equals single. If the column type equals single, control passes to block 2947, which puts the IEEE single precision key part value in arithmetic register #0 (described in detail in FIG. 42). Block 2948 then rounds the arithmetic register #0 to the size of the key buffer (described in detail in FIG. 41). Block 2949 then converts arithmetic register #0 and stores it into the key buffer (described in detail in FIG. 43). The procedure then exits normally.

Detailed Description Text (60):
FIG. 30 details the steps performed to seek to a record (see FIG. 28, block 2804). Predefined procedure block 3001 maps session and table IDs to the table instance and passes control to decision block 3002. Decision block 3002 determines whether the IDs are valid. If the IDs are not valid, the procedure exits with an error. If the IDs are valid, control passes to decision block 3003, which determines whether the table is a virtual table. If the table is a virtual table, the procedure exits with an error. If the table is not a virtual table, control passes to decision block 3004, which determines whether the table index pointer is null. If the table index pointer is null, the procedure exits with an error. If the table index pointer is not null, control passes to decision block 3005. Decision block 3005 determines whether the current key part number is zero. If the current key part number is zero, the procedure exits with an error. If the current key part number is not zero, control passes to decision block 3006. Decision block 3006 determines whether the seek mode is valid. If the seek mode is not valid, the procedure exits with an error. If the seek mode is valid, control passes to block 3007, which computes the number of real (not ordinal) key parts that have been set. Block 3008 then copies the real key parts into the record area. Block 3009 copies the ordinal column key parts into the subscript value table. I/O block 3010 positions the file using the current index and the key value in the record and passes control to decision block 3011. Decision block 3011 determines whether a record was found. If no record was found, the procedure exits with an error. If a record was found, control passes to block 3012, which positions to a row (with parameter value seek) (described in detail in FIG. 32A). Decision block 3013 determines whether the row was found. If no row was found, the procedure exits with an error. If the row was found, control passes to decision block 3014, which determines whether the Seek mode is greater than or equal to. If Seek mode is not greater than or equal to, the procedure exits normally. If Seek mode is greater than or equal to, control passes to decision block 3015. Decision block 3015 determines whether the key value in the key buffer equals the key value in the record buffer. If the key value in the key buffer does not equal the key in the record buffer, the procedure exits with a warning. If the key value in the key buffer does equal the key in the record buffer, the procedure exits normally.

Detailed Description Text (74):
If decision block 3235 determines that the parameter equals seek, control passes to predefined procedure block 3236, which normalizes the subscripts in the subscript value table. Control then passes to decision block 3237, which determines whether the key value in the key buffer equals the key value in the record buffer. If the key value in the key buffer equals the key value in the record buffer, control passes to decision block 3238, which determines whether the seek mode equals greater than. If the seek mode does not equal greater than, the procedure exits normally. If the seek mode equals greater than, control passes to predefined procedure block 3239, which increments the subscripts in the subscript value table. Control then passes to decision block 3240, which determines whether the subscripts have overflowed. If the subscripts have not overflowed, the procedure exits normally. If the subscripts have overflowed, control passes to predefined procedure

block 3241, which initializes the subscripts in the subscript value table and returns control to block 3232.

Detailed Description Text (75):
If decision block 3237 determines the key value in the key buffer does not equal the key value in the record buffer, control advances to decision block 3242. Decision block 3242 determines whether the seek mode equals equal to. If the seek mode equals equal to, control advances to decision block 3246, which determines whether the key permits duplicates. If the key does not permit duplicates , the procedure exits with an error. If the key permits duplicates, control passes to decision block 3247, which determines whether the mismatch in the key values occurred in a subscript (from an ordinal column). If the mismatch in the key values occurred in a subscript, the procedure exits normally. If the mismatch in the key values did not occur in a subscript, the procedure exits with an error.

Detailed Description Text (76):
If decision block 3242 determines that the seek mode does not equal equal to, control passes to decision block 3243, which determines whether the key value in the key buffer is greater than the key value in the record buffer. If the key value in the key buffer is greater than the key value in the record buffer, the procedure exits with an error. If the key value in the key buffer is not greater than the key value in the record buffer, control passes to decision block 3244, which determines whether the subscript values in the key buffer are greater than those contained in the subscript value table. If the subscript values in the key buffer are not greater than those contained in the subscript value table, the procedure exits normally. If the subscript values in the key buffer are greater than those contained in the subscript value table, control passes to predefined procedure block 3245, which initializes the subscripts in the subscript value table and returns control to block 3232.

Detailed Description Text (99):
Predefined procedure block 3534 allocates a temporary cell to hold the values of the keys numbered currKey and uniqKey and passes control to block 3535, which copies the value of the key numbered uniqKey from the regular record buffer to the temporary cell. Block 3536 then copies the value of the key numbered currKey from the regular record buffer to the temporary cell and passes control to decision block 3537. Decision block 3537 determines whether the key numbered currKey is unique-valued.

Detailed Description Text (100):
If decision block 3537 determines that the key numbered currKey is unique-valued, control passes to block 3538, which copies the value of the key numbered uniqKey from the temporary cell to the update record buffer. Then I/O block 3539 performs a random read into the update record buffer using the key numbered uniqKey, and control advances to predefined procedure block 3550.

Detailed Description Text (103):
Decision block 3544 determines whether the value of the key numbered uniqKey in the temporary cell equals the value in the update record buffer. If the value of the key numbered uniqKey in the temporary cell equals the value in the update record buffer, control advances to predefined procedure block 3550. If decision block 3544 determines that the value of the key numbered uniqKey in the temporary cell does not equal the value in the update record buffer, control passes to I/O block 3545, which reads the next record into the update record buffer using the current key of reference. Decision block 3546 then determines whether an I/O error has occurred. If an I/O error has occurred, control advances to predefined procedure block 3550. If decision block 3546 determines that no I/O error has occurred, control passes to decision block 3547, which determines whether the value of the key numbered currKey in the temporary cell equals the value in the update record buffer. If the value of the key numbered currKey in the temporary cell equals the value in the update

record buffer, control returns to decision block 3544.

Detailed Description Text (104):
If the value of the key numbered currKey in the temporary cell does not equal the value in the update record buffer, control passes to block 3548, which sets a flag to indicate that the next record has already been read (see FIG. 32C, block 3214). Block 3549 then sets the error code to indicate the record was not found, and passes control to predefined procedure block 3550.

Detailed Description Text (105):
Predefined procedure block 3550 deallocates the temporary cell holding the key values and passes control to decision block 3551, which determines whether an error occurred. If an error occurred, the procedure exits with an error. If an error did not occur, control passes to decision block 3552, which determines whether the count of terms in the table's predicate equals zero.

Other Reference Publication (1):
"DATAPLEX: An access to heterogenous distributed databases", Chin-Wan Chung, Communications of the ACM, v33, n1, p. 70(11) Jan. 1990.

CLAIMS:

1. A method of operating a computer system to process SQL requests to access 3GL application data stored in said computer system, by using a relational database representation of said 3GL application data, comprising:

generating an execution plan as a function of a SQL request and as a function of information provided in a predetermined catalog stored in said computer system, said predetermined catalog defining a relational database representation of underlying 3GL application data stored in said computer system; and

processing said SQL request within said computer system to access said 3GL application data, by executing said execution plan as a function of said information in said predetermined catalog and as a function of said underlying 3GL application data.

2. A method of operating a computer system to process SQL requests to access 3GL application data stored in said computer system, by using a relational database representation of said 3GL application data, comprising:

establishing a predetermined catalog by,

generating 3GL file schema from selected 3GL source programs that utilize said underlying 3GL application data;

specifying relational database schema as a function of said generated 3GL file schema; and

storing said 3GL file schema and said relational database schema in said catalog;

generating an execution plan as a function of a SQL request and as a function of information provided in said predetermined catalog stored in said computer system; and

processing said SQL request within said computer system to access said 3GL application data, by executing said execution plan as a function of said information in said predetermined catalog and as a function of said underlying 3GL application data.

3. The method of claim 2, said step of specifying said relational database schema,

comprising:

creating table definitions, creating column definitions, creating table predicate definitions, and creating table index definitions, all as a function of said 3GL file schema.

5. The method of claim 4, said extracting step comprising:

extracting file characteristics from said identified file definitions, said file characteristics including, file name, record length, key descriptions and file organization; and

extracting record structures from said identified file definitions, said record structures including data item offsets, data item lengths, and data item types.

7. A method of operating a computer system to process SQL requests to access 3GL application data stored in said computer system, by using a relational database representation of said 3GL application data, comprising:

generating an execution plan as a function of a SQL request and as a function of information provided in a predetermined catalog stored in said computer system, said predetermined catalog defining a relational database representation of underlying 3GL application data stored in said computer system, and said predetermined catalog stored in said computer system including 3GL file schema generated from selected 3GL source programs that utilize said 3GL application data, and including relational database schema specified as a function of said generated 3GL file schema; and

processing said SQL request within said computer system to access said 3GL application data, by executing said execution plan as a function of said information in said predetermined catalog and as a function of said underlying 3GL application data, said processing step comprising,

opening a table specified by said SQL request, including,

retrieving relational database schema from said predetermined catalog as a function of a table name in said SQL request,

retrieving 3GL file schema from said predetermined catalog as a function of said retrieved relational database schema, and

opening a 3GL application data file as a function of said retrieved 3GL file schema;

performing at least one action on said opened 3GL application data file, in accordance with an operation included in said SQL request; and

closing said table as a function of said retrieved relational database schema and said retrieved 3GL file schema.

8. The method of claim 7, said 3GL file schema including record structures of said opened 3GL application data file, and said relational database schema including column definitions of said relational database representation of said 3GL application data, said performing step comprising:

retrieving at least one data item from said opened 3GL application data file as a function of said record structures and said column definitions.

10. The method of claim 7, said 3GL file schema including record structures of said opened 3GL application data file and said relational database schema including

column definitions of said relational database representation of said 3GL application data, said performing step comprising:

storing at least one data item in said opened 3GL application data file as a function of said record structures and said column definitions.

14. The method of claim 7, said relational database schema including table definitions and table predicate definitions, said performing step comprising:

eliminating records within said opened 3GL application data file in accordance with said table predicate definitions, to create a subset of records in said opened 3GL application data file, said subset of records corresponding to said table definitions; and

performing said at least one action on said subset of records.

15. The method of claim 7, said relational database representation of said 3GL application data including tables, each table having at least one row, and said 3GL file schema including record structures of said 3GL application data that define at least one repeating data item, said performing step comprising, for each table:

creating said at least one row for each occurrence of said at least one repeating data item in accordance with said record structure and in accordance with said relational database schema.

17. A computer system for processing SQL requests to access 3GL application data stored in said computer system, comprising:

storage means for storing 3GL application data and for storing a predetermined catalog defining a relational database representation of said 3GL application data;

data input means for receiving a SQL request from a system user;

means for generating an execution plan as a function of said SQL request and as a function of information provided in said predetermined catalog; and

means for processing said SQL request to access said 3GL application data by execution said executing plan as a function of said information in said predetermined catalog and as a function of said 3GL application data.

18. A computer system for processing SQL requests to access 3GL application data stored in said computer system, comprising:

means for storing 3GL source code programs that utilize said 3GL application data;

means for generating a predetermined catalog defining a relational database representation of said 3GL application data, comprising:

means for generating 3GL file schema from selected ones of said 3GL source programs;

means for generating relational database schema as a function of said generated 3GL file schema; and

means for storing said 3GL file schema and said relational database schema as said predetermined catalog;

storage means for storing 3GL application data;

data input means for receiving a SQL request from a system user;

means for generating an execution plan as a function of said SQL request and as a function of information provided in said predetermined catalog; and

means for processing said SQL request to access said 3GL application data by executing said execution plan as a function of said information in said predetermined catalog and as a function of said 3GL application data.

19. A computer system for processing SQL requests to access 3GL application data stored in said computer system, comprising:

storage means for storing 3GL application data and for storing a predetermined catalog defining a relational database representation of said 3GL application data, said predetermined catalog including 3GL file schema generated from selected 3GL source programs that utilize said 3GL application data, and including relational database schema generated as a function of said generated 3GL file schema;

data input means for receiving a SQL request from a system user;

means for generating an execution plan as a function of said SQL request and as a function of information provided in said predetermined catalog; and

means for processing said SQL request to access said 3GL application data by executing said execution plan as a function of said information in said predetermined catalog and as a function of said 3GL application data, said means for processing, including,

means for opening a table specified by said SQL request, including, means for retrieving relational database schema from said predetermined catalog as a function of a table name in said SQL request, means for retrieving 3GL file schema from said predetermined catalog as a function of said retrieved relational database schema, and means for opening a 3GL application data file as a function of said retrieved 3GL file schema;

means for manipulating said opened 3GL application file, in accordance with an operation included in said SQL request; and

means for closing said table as a function of said retrieved relational database and said retrieved 3GL file schema.

20. The computer system of claim 19, said relational database schema including table definitions and table predicate definitions, said means for manipulating further comprising:

filter means for eliminating records within said open 3GL application data file as a function of said table predicate definitions, and for creating a subset of records in said opened 3GL application data file, said subset of records corresponding to said table definitions in said predetermined catalog; and

means for manipulating said subset of records as a function of said operation included in said SQL request.

Previous Doc Next Doc Go to Doc#

L11: Entry 5 of 5 File: USPT Jan 12, 1999

DOCUMENT-IDENTIFIER: US 5860136 A
TITLE: Method and apparatus for use of associated memory with large key spaces

Abstract Text (1):
To provide fast access times with very large key fields, an associative memory utilizes a location addressable memory and lookup table to generate from a key the address in memory storing an associated record. The lookup tables, stored in memory, are constructed with the aid of arithmetic data compression methods to create a near perfect hashing of the keys. For encoding into the lookup table, keys are divided into a string of symbols. Each valid and invalid symbol is assigned an index value, such that the sum of valid index values for symbols of a particular key is a unique value that is used as an address to the memory storing the record associated with that key, and the sum of keys containing invalid index values point to a location in memory containing similar data. Utilizing the lookup tables set and relational operations maybe carried out that provide a user with a maximum number of key records resulting from a sequence of intersection, union and mask operations.

Brief Summary Text (13):
Traditional approaches for designing a network address structure have either been intimately entwined in the design of efficient routing look-up tables or assigned by a central authority such as ARPANET. Neither of these approaches gives much if any thought to the needs, desires or ease of use of the group which must make operational use of the system. In an age of fourth generation database languages and high level compilers, network addresses are basically hand-coded in low level language. Addresses and address structures are difficult to change as a mobile end-unit moves from one communication network to another. Experts are often required to ensure that operational equipment is properly integrated into the system. ISO (International Standards Organization) addressing provides a basis for a much better approach but the overall design and administration of a network addressing structure must be elevated to an easily supported, user friendly, distributed architecture to effectively support the user's long-term needs.

Brief Summary Text (14):
Traditional directory access methods, whether for Internet routing, databases or compiler symbol tables, fall into three basic categories:

Brief Summary Text (20):
Hashing has often been used over the last several decades to create directories where fast access is desired. One system uses a multi-level hashing scheme as the file system directory structure. The Total database system is based on hashed key access. Many language compilers use hash tables to store symbols. Hash table schemes have good average access costs--often a single access, but can degrade drastically when the table becomes too full or the hashing function does not perform a good job of evenly distributing the keys across the table. Some techniques called "linear hashing" and "dynamic hashing" have provided the method of expanding the hash table when a particular bucket becomes too full instead of using the traditional linked list overflow methods. These techniques generally

require about 40% more space than the number of active addresses (keys) to achieve single access speed without employing overflow methods.

Brief Summary Text (43):
Another aspect of the invention is an apparatus and method for implementing a routing table directory to provide for fast access times to look up routing information. This apparatus is an application of a novel associative memory utilizing arithmetic coding to associate a key presented to the memory with a record stored in the memory, but has a very-wide range of application in many different types of data processing systems. The associative memory includes an index table stored in memory and a record memory for storing the records of data. The index table is constructed such that each symbol of a key, a key being divided into a string of symbols and each symbol being defined by its position within the key and its value, addresses an index value in the index table memory. These index values are assigned such that the sum of index values for a given key is a unique value that is used to address the record memory. Several methods and apparatus are disclosed the permit random assignment of index values to new keys as they are presented, as well as for keys that are presented in sorted order for addition to the memory.

Brief Summary Text (44):
Another aspect of the invention provides a method and apparatus for utilizing use-count tables created by the arithmetic coding process to determine the maximum number of key sets resulting from the set operations union and intersection, used to combine two or more different key sets. The intersection of the key for two or more relational database tables is essentially the relational join operations. This method can perform the relational join operations in a much faster and efficient method than presently utilized joined operations.

Drawing Description Text (12):
FIG. 10 is a flow diagram describing the operation of a change old index value component of the add key logic of FIG. 8.

Drawing Description Text (16):
FIGS. 14A and 14B are a flow diagram describing a method for assigning index values to an entire key set.

Drawing Description Text (17):
FIGS. 15A, 15B and 15C are a flow diagram describing another method for assigning index values to an entire key set.

Drawing Description Text (24):
FIG. 22 is a flow diagram illustrating the method for assigning index values to an entire key set group, including all invalid symbol values.

Detailed Description Text (66):
Thus there has been disclosed a data communication system which uses a routing table access method that treats network addresses as variable length symbol strings without internal structure--i.e., as flat addresses--to simplify the handling of mobile end-systems simultaneously connected to multiple access points. The system utilizes high speed, Media Access Control and Internet processes which handle multicast messages to multiple, mobile hosts. The technique is also applicable to real time database applications such as a network name service which relates a logical name (alphanumeric name) to its universal identification code. For example, an automatic telephone directory service could use this system to enable entry of a particular name and receive the telephone number of that name. Thus, the novel system allows one entity having a universal identification number to communicate with any other entity in the system having a universal identification number but whose physical location is unknown. Because the Internet router system is based on a flat logical address space, it provides efficient routing of both multicast and

unicast packets independent of the internal network address format or structure.

Detailed Description Text (69):
The routing table access method and apparatus described in connection with FIGS. 2-4 has, as already discussed, real time data base applications other than the date communications network of FIG. 1. Those of ordinary skill in the art will readily recognize that the routing table access method and apparatus of FIGS. 2-4 describe an associative memory employed in context of a communications switching application. The possible applications of this same associative memory scheme are numerous and diverse; it is not confined to communications switching systems. For example, it has application in such diverse systems as those for on-line telephone directories, radar target tracking, and sonar signal classification--almost any system or application requiring or using high speed access to real-time databases where information is accessed with key values without knowledge of precisely where it is stored within a memory system. It is especially useful for systems utilizing very large key-spaces, when the number of keys that are actually used to access data records stored in memory constitute a fraction of the total number of possible keys. For these reasons, the two additional methods of constructing the index table will be described with reference to a generic application in a host system.

Detailed Description Text (72):
A key symbol, when received, is stored in key symbol buffer 104. Symbol counter 106 counts the symbols as they are received so that the position where the current symbol is stored in the key symbol buffer 104 is always known. The value of the symbol counter is the current symbol position "i". As represented by block 503, an address for index table 68 is generated from the position "i" and the value of the key symbol stored in the buffer 104. The position of the symbol within the key taken from the symbol counter 106 selects a bank, Bank[i], in key index table memory 68, and the value of the key symbol taken from buffer 104, Symbol[i], is the offset address within the bank.

Detailed Description Text (73):
Key index table memory 68 basically stores a table of values, called key index values. The memory storing this key index table must then be divided, either physically or logically, into banks 1 to N, as shown. Each bank is, in turn, divided, either physically or logically, into offset addresses that a predetermined number of bits that store a key index value. The size or number of bits addressed by the offset must be large enough to accommodate the size of index values stored therein, and the number of offset addresses depends on the size of the key symbols. The number and size of offset addresses depends entirely on the application.

Detailed Description Text (75):
The data value or index that is addressed is read onto line 505 to arithmetic computation logic circuitry 72. Arithmetic computation logic is primarily comprised of a Modulo (P) adder, where P is approximately the number of logically addressable memory locations in key record memory 78. In the preferred embodiment, P is chosen to be a prime number. Arithmetic computation logic circuitry 72 is initialized or set to zero before receipt of the first symbol in a key is presented on line 501. As the index value is read onto bus 505 from the index table memory, symbol counter 106 provides an enabling or clock signal on line 507. The enabling or clock signal from the symbol counter is delayed by delay device 509 by a time greater than the access time required for the index table memory 68 but less than the period between symbols presented on input bus 501. When enabled, the arithmetic computation logic circuitry adds an index value on line 505 to a previously computed sum, in effect keeping running total. When the running total exceeds P, P is subtracted from the running total. The arithmetic computation logic circuitry thus performs a Modulo (P) addition of the index values stored in the index table memory for each symbol in the key to create a final sum called a record index. The record index is a data value that will be used as a logical address to the place within key record memory 78 in which the record corresponding to the key presented on input lines 501 is

stored.

Detailed Description Text (76):
Arithmetic computation logic circuitry also includes zero detect circuitry 90 for indicating that an index value received on line 505 from the index table memory 68 is zero. By definition, a zero value stored in an entry the index table for a symbol, as defined by its position within the key and its value, indicates that the symbol value has not been encoded into the index table and therefore no key record is stored in the key record memory 78 associated with the key that has been presented. The zero detect or "No Index" signal is provided on line 92 to the host system.

Detailed Description Text (78):
Once the last symbol of the key is processed, the final sum or record index in the arithmetic computation logic circuit 72 is read to the record index buffer 74, also called a key record memory address buffer. This value is the key record memory address that is presented on address lines 517 to key record memory 78 in order to access the record associated with that key. An enabling or clocking signal on line 513 causes the record index buffer 74 to store the record index and provide it on address lines 517 to the key record memory 517. To generate the enabling or clocking signal on line 513, the enabling signal provided to arithmetic computation logic circuit is divided by N, the number of symbols in the key, by divider circuitry 515.

Detailed Description Text (79):
Record memory 78 stores records of data that are accessed by presenting a key to the associative memory module 500 and decoding it into a record memory address as described above. Upon presentation to the record memory, the record memory address enables access of a record associated with the key and, with an appropriate read command (not shown) reads it onto output lines 519 to be stored by key record buffer 80 when enabled by a signal from delay element 515 (the delay element providing sufficient time for accessing the memory and reading out the record of data onto lines 519).

Detailed Description Text (86):
Adding records and their keys requires that its key be "learned". The key for the record to be added is presented to the associative memory module on input lines 501; the record to be added is placed in key record buffer 523; and an "ADD RECORD" command is given on line 517. Each symbol of the key is processed as previously described. Frequently, at least one, and sometimes all but one, of the symbols in the key has already been encoded. If a symbol (as defined by its value and position within the key) has been encoded, learned key logic performs no function, and the key index values are read into arithmetic computation logic circuitry 72. However, when zero detect circuitry 90 reads a zero on line 505 from the key index memory 68, a zero detect signal is provided on line 92 to learned key logic 88, as well as to the host system as a "No Index" signal. This "no index" or zero detect signal means that the table entry--the symbol position and value--contains a zero. At this point, learned key logic circuitry 88 generates a new key index value and provides it to key index table 68 on write index bus 520. The index table memory then stores it in the entry of the key index table memory indicated by table address 503. As will be discussed in connection with the remaining Figures, learned key logic circuitry 88 requires, in order to generate the new index value for the key symbol, the key symbol value and position, and index values on read index line 505. Therefore, learned key logic is coupled to the key symbol buffer 104, the symbol position counter 106, and read index bus 505.

Detailed Description Text (87):
Processing the key symbols presented on lines 501 then continues as before, with learned key logic generating new key indexes as necessary. When all of the symbols of the key have been processed, and their corresponding key index values added to a

record index value 74 by the arithmetic computation circuitry 72, the record in buffer 523 is stored in the key record memory 78 at the location indicated by the record index.

Detailed Description Text (88):
To delete records, the host system presents a "DELETE RECORD" command on line 519. To perform the delete function, the key must be presented on input key bus 501. Learned key logic 88 then deletes any key index value that is being used only by the key associated with that record by writing a zero into the table entry with write index bus 520. Deleting any unused index values permits them to be reused for the un-encoded key symbols that may be subsequently presented, thereby providing more efficient use of the record memory 78. The "deleted" record is simply overwritten with a new record when one is presented.

Detailed Description Text (89):
Please note that where an ability to add, delete and update records is not desired, learned key logic circuitry is not necessary. All that is required is an encoded index table memory 68, arithmetic computation circuitry 72, key record memory 78 written with all of the records and suitable timing circuits and buffers. The index values that are stored in the index table memory 68 may be generated separately on a general purpose digital computer, from a known set of keys, and then stored in the memory. One of these methods for doing so has already been described in connection with FIGS. 2-4. Other methods will be described in connection with the remaining figures. If the index table memory is a ROM, for example, the index values are stored in the manner provided by the ROM device.

Detailed Description Text (91):
The function of add key logic circuitry is to generate new keys that are provided to write index bus 520 for storing in the key index table memory 68 (FIG. 5). In one embodiment, add key logic recycles key symbols to key symbol buffer 104 with bus 609. Add key logic is coupled to symbol counter 106, key symbol buffer 104, read key index bus 505, zero detect line 92 and "ADD RECORD" command line 517 so that it receives the current key symbol position or count, key symbol value and key index in addition to the zero detect and add command. Add key logic provides an increment (INC) command on line 607 to symbol use count logic 603 for each key symbol presented on line 501 during an add key operation.

Detailed Description Text (92):
Symbol use count logic 603 tracks for each key symbol value and position, or entry in the key index table memory 68, the number of different keys that share that particular symbol value and position. Symbol use count logic receives the symbol value and symbol position from key symbol buffer 104 and symbol counter 106, and the increment command signal from add key logic on line 607. A key index value for a particular symbol that is shared by more than one key cannot be deleted. If symbol use count logic circuitry indicates that a particular symbol has been encoded into the index table for only one key, then delete key logic circuitry 605 writes a zero to the write index bus 520.

Detailed Description Text (93):
Referring now to FIG. 7, symbol use count logic circuit 603 tracks the number of times a particular symbol, Symbol[i] in a particular symbol position or bank i, is used by different keys. The key index value assigned to Symbol[i], INDEX[i,Symbol [i], can be deleted when it is no longer being used by any key. Essentially, it uses a symbol use count table memory 801 that is constructed and addressed like the key index table memory 68 (FIG. 5). As indicated by summer 802, values in symbol counter 106 and key symbol buffer 104 are used as bank and offset addresses, respectively, to a particular table entry for the key currently being processed. The data value stored in each entry is the number of keys that are currently encoded that share or use the value Symbol[i] in position i.

Detailed Description Text (94):
When the table memory 803 is addressed, the use count for Symbol[i] is read into counter 803. When a key is being added, the value in counter 803 is then incremented by one with an INC command signal on line 707 from an add key logic circuit. In essence, this counts the number of times a particular symbol[i] has been presented to the associative memory as part of a key that is being added. When a record is being deleted, counter 803 is decremented by one with the delete record command on line 519.

Detailed Description Text (96):
Referring now to FIGS. 8 to 13, these figures disclose a third method (in addition to the two methods previously described in connection with FIGS. 2-4) of adding keys (the actual records may be stored later) using a different add key logic circuit. Basically, this method or process assigns key index values for key symbols when keys are presented in a monotonically sorted or lexigraphical order (either ascending or descending) for entry into the associative memory. The sorted order of the keys helps to eliminate "holes" for possible combinations of key symbols which are not presented in a sorted order or sequences, creating a near "perfect hashing" or "perfect packing". Should a key be presented whose sum of key index values is the same as the sum of key index values for a previously entered key, current or previous key index value assignments are changes, and any previously stored key records are moved in memory to create a "hole" for the new key. The adding process is completed when the highest or last record memory location is filled.

Detailed Description Text (97):
Referring now to FIG. 9 only, showing in schematic representation a dedicated circuit embodiment utilizing this method, add key logic circuitry 601 includes as major components: save table 901; change table 903; create index values logic circuit 905; change old value logic circuit 907; save new index logic circuit 909; and a directory table 911. The save table 901, change table 903 and directory table 909 are data structures stored in a random access memory. The logic circuits 905, 907 and 909 are shown to be dedicated circuits (possibly LSI or VLSI devices), but may implemented with programmed logic devices, general purpose computers, a microprocessor, or a combination of these performing the steps of the method.

Detailed Description Text (98):
Both the save table and the change table have one row of values for each symbol position i in a key, i=1 to N, as shown by symbol position columns 913 in each table. Symbol counter 106 is therefore used to select a row in the save table, identified as Save[i], and in the change table, identified as Change[i]. Each table has three additional columns for storing values associated with Save[i] and Change [i]. The save table keeps values of the symbols for the current key being added.

Detailed Description Text (99):
In the save table 901, column 915 stores a value Save[i].symbol. This value is set equal to the symbol value, Symbol[i], from symbol buffer 104. Thus, Save [i].symbol=Symbol[i]. Column 917 receives from the key index table on read key index bus 505 an index value for Symbol[i], Index[i,Symbol[i]], and stores it. The column 919 receives the zero detect logic signal on line 92 and stores a value for a variable Save[i].new. This value is "new" (actually a data value arbitrarily chosen to represent "new") when a zero detect signal is received, indicating that the value for Save[i].index is new and that Symbol[i] had not been previously encoded into the key index table memory. Thus, Save[i].new=new. Otherwise, a data value representing "old" is stored: Save[i].new=old.

Detailed Description Text (101):
The change table 903 keeps information on the last index values assigned for each symbol position to allow "holes" to be made for new keys and to move any records if required.

Detailed Description Text (102):
The change table 903 stores the previous symbol, record index and key index values for each symbol position [i] that are need to generate the next key index values for the next new key presented to the associative memory. Change table 903 therefore has three columns, one for the last assigned values of the following variables or data elements: for the record index value, Change [i].record; for the key index value, Change[i].index; and for the symbol value, Change[i].symbol. Like the Save table, the symbol position counter selects the row in the table, enabling entries on that row to be accessed (for reading and writing) by bus 922.

Detailed Description Text (103):
A directory table 923 is also kept. A row of the table is selected with a data value generated by create index values logic 905 and save new index logic 909. This value is notated simply as "sum", as it is a sum of all of the key index values for a key that is being processed. The number of rows in the table equals the number of record locations in record memory 78. The value "sum" is in fact equivalent to the record index value for the key being processed. Each row of the table has one entry, Change[sum].dir, for each record location in the record memory, sum=1 to P. This entry has one of two values; one representing the location is "assigned" and the other indicating the location is "unassigned". This data structure may be set up in any memory element in the associative memory or host system. It can even be made part of the record memory 78 (FIG. 5).

Detailed Description Text (104):
Coupled between buses 920 and 922 are create index logic 905, change old index logic 907 and save new index logic 909 which permit the logic elements to read and write values to the respective tables. Please note, however, the buses 920 and 922 are merely a functional representation, intended to simplify presentation. The actual data exchange structures between the different elements of add key logic 601 may differ substantially depending on the memory and logic elements selected to implement the data structures of the tables and the logic. For example, when implemented on a specially programmed general purpose digital computer, the data exchange structures will be those of the particular computer chosen.

Detailed Description Text (105):
Save new index logic 909 is further coupled to the write key index table bus 520 so that new index values can be stored in the key index table memory 68, as well as to key symbol buffer for recycling key symbol values.

Detailed Description Text (107):
At start-up of the associative memory, as indicated by circle 1001, all logic components are reset at step 1003, and all tables initialized to zero (written with all zero values) at step 1005. Step 1005 includes setting Index[i,Symbol[i]] to zero, for all i and Symbol[i]. When the host system presents a new key for addition or encoding into key index table memory 68 (FIG. 5), it signals at step 1007 that a new key is being presented on key symbol bus 501 (FIG. 5) and the create new index value process begins. Step 1009, (FIG. 9) indicates when each symbol within the key is presented for processing as indicated by the symbol count being greater than zero. When a symbol is presented, the create new index value process continues to decision step 1011. If the new symbol is the first symbol of the key, then Sum is set equal to zero and i equals one. Further, a variable Any.sub.-- New is set equal to "old". Any.sub.-- New is subsequently set to "new" if any Save[i].new="new" from i=1 to N. In other words, "new" is entered in the new column 919 for any symbol position in the Save table; otherwise it is "old". Any.sub.-- New is a flag which indicates that one or more of the Save [i].new values is set to "new".

Detailed Description Text (108):
At decision block 1015, if the key being presented to the associative memory is to be added, as indicated by the Add command signal on line 517 (FIG. 6) being turned on, create index logic performs the steps outlined in block 1017. If the key is not

being added, the created index value process ends. The process steps of block 1017 involve setting the Save table entry Save[i].symbol equal to the symbol value Symbol[i] from the key symbol buffer 104 (FIG. 6).

Detailed Description Text (111):
Referring now to FIGS. 9 and 11, if the sum of the record index values is a record index value addressing a location in record memory 78 (FIG. 5) already having a key assigned to it, then this new key cannot be entered until either the previous key is moved to a new location or the new key's sum is increased until an unassigned location is found. Either of these moves are accomplished by increasing the value of an index value assigned to some previous key and also used by the new key being processed or increasing the value of an index newly assigned to this key.

Detailed Description Text (112):
First, change old index logic 907 checks to see if the sum of the Index values for the new key, the sum of Index[i,Symbol[i]] from i=1 to N has been assigned. To do this, change old index value logic gets the value for Sum for create index logic 905 on bus 922. As indicated by decision block 1103, change old index value logic selects from the directory table 923 using the select line 925 the value for Directory[sum], which then reads this value out onto bus 922 for retrieval by change old index logic. If Directory[sum] equals "unassigned", this indicates that the index values for the new key sum to a record index that has not previously been assigned to a key. Change old index logic then determines whether Any.sub.-- New="new", as indicated by decision block 1105. If Any.sub.-- New="old", the process ends. If Any.sub.-- New="new", then save new key logic 909 is signalled to begin and the value for the variable Sum passed to it, as indicated by blocks 1107 and 1109, and the process carried out by change old index logic ends.

Detailed Description Text (113):
If, on the other hand, Directory[sum] is assigned, the process of change old index logic continues at decision block 1111. If Any.sub.-- New="old", a previously assigned index value must be increased to make a "hole" for the new key. The process proceeds to the steps shown in FIG. 12, which are discussed in connection with that figure.

Detailed Description Text (114):
Otherwise, if Any.sub.-- New="new", indicating that one of the index values for the current key was newly assigned or created by create index value, this new index value is incremented by one, thereby incrementing Sum until an "unassigned" location in the Directory table, Directory[sum]="unassigned", is found. An "unassigned" value indicates that the that a record memory location addressed by the a record index value equaling Sum has not been assigned to a key and is available for storing a record associated with the new key.

Detailed Description Text (116):
Referring now to FIGS. 9 and 12, where Directory[Sum]="assigned" and Any.sub.-- New="old" at steps 1103 and 1111 in FIG. 11, the processes of change old index logic 907 carry on by increasing a previously assigned index value in order to make a "hole" in the record memory for the record associated with the new key. Three additional variables are required to continue the change old index value process. Directory.sub.-- Position is a pointer to the Directory table 923 that will point to the record memory location at which the hole is made. Symbol.sub.-- Position keeps track of the symbol position of the index value which will be increased to make a hole. As indicated by block 1201, Directory.sub.-- Position is initially set equal to Sum, and Symbol.sub.-- Position is set equal to 0.

Detailed Description Text (117):
Starting with block 1203, a loop begins that is designed to find an index value in the change table 903 that was previously assigned to one of the symbols of the key currently being processed. To do this, the loop begins at i=1 and compares Change

[i].Symbol in the change table and Save[i].symbol in the save table to see if Change[i].Symbol=Save[i].Symbol, as shown in block 1205. Where Change [i].Symbol=Save[i].Symbol, two other conditions are tested. First, Change [i].record, the record index value or Sum at the time the Symbol[i] was assigned, is compared against the current Sum, as shown in block 1207. Second, it is compared with Directory.sub.-- Position, as shown by block 1209. If Change[i].Symbol is greater than both, then: in block 1211, Symbol.sub.-- Position is set equal to i and Directory.sub.-- Position is set equal to Change[i].Record; and, in block 1213 and decision block 1215, i is incremented by one and the loop repeated until i=N. Otherwise, the steps of block 1211 are not performed and the loop repeated for i=i+1 until i=N. In effect, this loop is not only trying to find an index value in the save table having a symbol shared by the current or new key, it also finds, where there is more than one such symbol, the index value that, when assigned, had the largest record index value. The Directory.sub.-- Position variable keeps track of the largest record index value found as the loop is performed.

Detailed Description Text (120):
Referring now to FIG. 13, once the processes of FIG. 12 are complete, change old index logic then signals store record logic 94 on line 95 (FIG. 5), as indicated by block 1301, and passes to stored record logic the values for Directory.sub.-- Position, Increase and Last.sub.-- Assigned (a value is assigned to this variable by save new index logic as shown in FIG. 14). Store index logic utilizes these values to move a block of records within the record memory 78 (FIG. 5) addressed by record index values between Directory.sub.-- Position and Last.sub.-- Assigned to new a new block of locations addressed by record index values from Directory.sub.-- Position+Increase to Last.sub.-- Assign+Increase, thereby making the "hole" in the record memory for storing the record associated with the new key.

Detailed Description Text (127):
Referring now to FIGS. 3, 8 and 15, there is illustrated the hardware and a flow diagram required for the assigning of index values to an entire key-set group using an alternative method for assigning index values. The use-count table (801 of FIG. 8) is updated as a set of keys are presented to the system. Before the process of updating the use-count table begins, all positions within the use-count table are reset to zero or a null value at step 1500. The first key is then presented at step 1502 to the system logic and the first symbol of the key is read at step 1504.

Detailed Description Text (130):
An inquiry at step 1540 determines if another symbol value exists for the present symbol position. If so, the CURRENT COUNT is incremented by one at step 1542 and control returns to step 1517. When no further symbol values exists in the current bank, the control logic looks for the next bank at step 1546. If another bank exists, the BASE value is set equal to the present BASE value times current count at step 1548 and control returns to step 1514. Otherwise, all index values for the keys have been assigned and the process is complete.

Detailed Description Text (131):
Referring now to FIGS. 8, 16, and 18 another process is illustrated for assigning index values to symbols for an entire key set. Initially all index-value tables and use-count values are reset to zero or a null value at step 1600. This includes initializing values in the maximum suffix table 1910 to the minimum suffix symbol string and initializing the minimum suffix table 1920 values to the maximum suffix string. Next, a key is presented at step 1602 to the control logic and the first symbol of the key is read at step 1604. The use count for the read symbol value at the present symbol position is incremented at step 1606 to indicate a use of the symbol value at the symbol position.

Detailed Description Text (132):
An inquiry is made at step 1608 to determine if the suffix of the present symbol is greater in the collating order than the stored MAX SUFFIX. The suffix of a symbol

in a key is the key ordered sequence of all the symbols of lower collating value than the current symbol. For example, the suffix of 3 in the key 12345 is 45 and the suffix of B in key ABCDE is CDE. The suffix relative size determination is based on the collating order of symbol in the suffix and not the index value. If the current suffix exceeds the previously stored MAX SUFFIX in collating order, the MAX SUFFIX is set equal to the present suffix at step 1610 and stored in the max suffix table 1910 (FIG. 18).

Detailed Description Text (133):
Should the suffix be less than MAX SUFFIX in collating order or after MAX SUFFIX is set equal to the present suffix, control passes to step 1612 where an inquiry is made to determine if the present suffix is less than the previously stored MIN SUFFIX symbol string. If so, at step 1614 the MIN SUFFIX is set equal to the present suffix symbol string and stored in the min suffix table 1920 (FIG. 18). Otherwise, step 1616 determines if another symbol value exists within the present key. If another symbol exists, the next symbol is presented to the control logic at step 1604; if not, step 1618 determines if another key must be presented to the control logic. Additional keys return control to step 1602.

Detailed Description Text (134):
Once all keys have been presented to the use count table 801 of 8 and 18, max suffix table 1910 of FIG. 18, and min suffix table 1920 of FIG. 17, control passes to step 1620 to begin the process of assigning index values to the key symbols. The use count table 801 of FIG. 7 and FIG. 18 stores integer count values. The max 1910 and min 1920 suffix table memories store symbol sequences. At step 1620, the base value for the least significant symbol position is set equal to one and the value for LAST NON-ZERO ENTRY is set equal to zero. Next at step 1622, the value for CURRENT COUNT is set equal to one. The least significant bank in the use-count table is scanned at step 1624. The initial symbol value in the bank is read at step 1626 and an inquiry is made at step 1628 to determine if the first symbol value use-count number is greater than zero. If not, control returns to step 1626 and the next symbol value in the bank is scanned. Otherwise, an inquiry is made at step 1630 to determine if the LAST NON-ZERO ENTRY value is greater than zero. If not, the symbol value is assigned an index value of one at step 1632, LAST MAX SUFFIX is set at step 1633 equal to a null string, the CURRENT COUNT is incremented at step 1634 and the LAST NON-ZERO ENTRY value is set equal to one at step 1636. Control then passes back to step 1626 and the USE COUNT value of the next symbol value in the use-count table is read.

Detailed Description Text (135):
If inquiry step 1630 determines the last non-zero value is greater than zero, the symbol position is assigned at step 1632 an index value according to the following equation: ##EQU1## The max suffix and min suffix index values are equal to the sum of the index values of the key symbols comprising the suffix. The CURRENT COUNT is incremented and the LAST NON-ZERO ENTRY value is set equal to the present index value at step 1634. At 1643, the last max-suffix value is set equal to the max-suffix value for the current position. The max and min suffix values for the least significant symbol position or bank are both zero.

Detailed Description Text (137):
Referring now to FIGS. 7, 16 and 19 there is described a method for incrementally updating the index tables and directory table of the present invention. This dynamic directory sizing process allows the number of non-zero entries in the index value tables to expand and contract as new keys are added or deleted and to minimize the size of the directory. This method expands the ADD KEY and DELETE KEY logic (601 and 605 of 88 in FIG. 6) processes earlier described. The USE-COUNT logic (603 in FIG. 6 and 7) remains the same. As before, the ADD-KEY logic will only operate when the ZERO DETECT (92 of FIG. 6) indicates that a symbol value location in a bank of the index value table has no index value assigned or the compare keys (2009 of FIG. 19) indicates a key mismatch. When the ZERO DETECT or

key mismatch is indicated, the ADD KEY logic 601 activates the ADD-NEW-SYMBOL process. The ADD-NEW-SYMBOL process produces a non-zero index value for a symbol value which previously had a zero or null index value or in which the count flag was zero. In addition, the ADD-NEW-SYMBOL process increases the higher order index values above the new index value.

Detailed Description Text (139):
Next, the partition point within the bank where the new symbol value will be added is determined at step 1702. When count flags are used, the compare keys logic 2009 of FIG. 19 compares the new key with the stored key and provides the learned key logic 88 with the symbol positions and symbol values that are different between the two keys. One of the symbol positions is selected and the symbol value having a count flag equal to zero has its count flag set to one. At least one of the two symbol values must have its count flag set to zero or the symbols would not have the same index value.

Detailed Description Text (143):
At step 1710, the key values in each of the NEW SIZE locations just below the block in the old directory moved at either step 1709 or 1712 are examined to determine if the symbol value at the partition point is greater than the PARTITION SYMBOL value. If the stored key symbol has a symbol value greater than PARTITION SYMBOL, the record associated with the key symbol is moved up from its current location in the old directory to the corresponding point in the NEW SIZE hole below the block just moved at either step 1709 or 1712. If the symbol value is equal to or less than PARTITION SYMBOL the record is left in its present position in the old directory. This process creates a hole below the previously moved block with selected records moved into the hole in the new directory table at step 1710. The hole size is equal to NEW SIZE. This hole accommodates entries using the new symbol value, including those which already used that value if count flags are used. At the end of the hole, another block of directory locations are added to the new directory at step 1712. The size of this block equals BLOCK SIZE. This next block to be moved starts at the next location below the last location of the previous block moved from the old directory to the new directory by either step 1709 or 1712.

Detailed Description Text (147):
Referring now to FIG. 17, once a use of a symbol value discontinues at step 1800, the use count for that symbol value is decremented by one at step 1802. A determination is then made at step 1804 to determine if the use count for the symbol value equals zero. If not, the process stops and no further action is taken. Should the use count equal zero, then all the other use counts using this index value must also be zero, before the index value may be removed. In other words, all symbol values with the same index values must all have zero use counts before the REMOVE KEY LOGIC begins a compact process to decrease the size of the directory table and reassign index values in the index-value table.

Detailed Description Text (152):
Referring now to FIGS. 19 to 21, there is illustrated yet another embodiment of the invention. The embodiments in FIGS. 19 to 21 are slightly altered from the apparatus disclosed in FIGS. 5, 6 and 7 and enables index values to be assigned to valid and invalid key symbol locations within the INDEX VALUE TABLE 68. This allows an index value to be calculated for record keys containing invalid key symbols. The calculated index value will point to a location in the neighborhood of index values for record data with similar key values.

Detailed Description Text (154):
The memory in-use logic 2002 checks a memory in-use flag bit within the key record memory 78 to determine if record data is stored in the location pointed to by a record index 74 value. The bit is set to one (1) if data is currently stored at a location and zero (0) if no data is currently stored at the location. The flag count logic 2003 is the value from the count flag logic 2000 generated when the

record index 74 is computed. In addition, the input key 501 being presented to the associative memory must be compared in compare keys logic 2009 to the sequence stored in the key record to determine if the keys are identical. If the input key 501 and the stored key 2010 are not identical, the learn key logic 88 is notified by the keys not equal flag line 2011.

Detailed Description Text (157):
Referring now to FIGS. 22a and 22b, there is illustrated a flow diagram describing the method for assigning index values to an initial or entire key-set group including valid and invalid symbol values using an alternative method for assigning index values. The USE-COUNT TABLE (801 of FIGS. 7 and 21) is updated as a set of keys are presented to the circuitry. Before the process of updating the USE-COUNT TABLE 801 begins, all positions within the USE-COUNT TABLE are reset to zero or a null value at step 2300. The first key (for instance a text string or DNA sequence) is then presented at step 2302 to the system logic and the first symbol of the key is read at step 2304.

Detailed Description Text (160):
Inquiry step 2340 determines if another symbol value exists for the current symbol BANK. If so, a determination is made at step 2318 if the SYMBOL VALID FLAG entry equals zero. If the entry equals zero, control passes back to step 2317 and the next symbol position for the present BANK in the USE-COUNT TABLE is read and assigned an index value which is the same as the previously assigned index value. If the count flag equals one, CURRENT COUNT is incremented by one at step 2342 and control returns to step 2317. When no further symbol values exists in the current BANK, the control logic determines if another BANK exist at step 2346. If another BANK exists, the BASE value is set equal to the present BASE value times current count at step 2348 and control returns to step 2314. Otherwise, all index values for the keys have been assigned and the process is complete.

Detailed Description Text (161):
Once completed, each symbol position within the INDEX VALUE TABLE 68 is assigned an index value, including invalid symbol positions not used by the presently stored symbol key set. With all the index table locations assigned a value, one can rapidly determine the "closeness" of an input key string sequence containing invalid symbols to key sequences already encoded into the USE-COUNT TABLE. Once the initial INDEX VALUE TABLE 68 values have been assigned using the process of FIGS. 22a thru 22c, additional keys may be added to associative memory. If the sum of the symbol index values identifies a location in RECORD MEMORY 78 where the IN-USE flag is not set (location not currently used) then the IN-USE flag is set and the record and key are stored with no modification to any index values in the INDEX VALUE TABLE 68. The USE-COUNT TABLE is updated with counts for all the new key symbols. If RECORD MEMORY 78 location for the new key (i.e., the sum of the symbol index values) is IN-USE (IN-USE set) and the keys are different then the values in the INDEX VALUE TABLE 68 must be modified to make room for the new key using the method described in FIG. 16.

Detailed Description Text (162):
When a key is entered for which the sum of the symbol index values equals a record location which is already in use, and the key already stored in the location is different from the new key, the values in the INDEX VALUE TABLE 68 must be expanded to make room for the new key (or one of the keys must be discarded). The new key and the stored key are compared in the compare keys logic 2009. The compare keys logic 2009 informs the learned key logic 88 of the symbol positions and symbol values that differentiate the two keys. Symbol positions with different symbol values must be assigned the same index value or the two keys would not have the same index sum. To distinguish the two keys, one of the different symbol values must be assigned a different index value. The process defined in FIG. 16 describes the method for assigning one symbol a new-index value and relocating all the effected key records.

Detailed Description Text (164):
The associative set processor 2400 is capable of performing a number of set operations. These operations include a union function 2406 for creating a table of all elements existing within the two USE-COUNT TABLES (2402 and 2404), an intersect function 2408 for creating a table containing all common elements between the two USE-COUNT TABLES and a mask function 2410 for combining a mask table, where some or all of the entries equals one (1), with a symbol USE-COUNT TABLE to create an output table, wherein each USE-COUNT position, having a value greater than zero (0) and the corresponding location in the mask table is equal to one (1), then the resulting table location is set to the USE-COUNT. The associative set processor 2400 outputs to a result table location 2412. A table access counter 2414 allows the associative set processor 2400 to sequentially read through all the locations of the two input tables 2402 and 2404 and place the results of the set operation in the corresponding location of the result table 2412. The table access counter includes the BANK and COUNT of FIG. 24. At the end of an operation on two input tables, set A and set B, the maximum set size 2415 has been computed for the resulting set C which is the largest number of records in the set resulting from the same union, intersect or mask operation being performed on the two key sets represented by table A 2402 and table B 2404.

Detailed Description Text (165):
Referring now to FIG. 24, there is shown a flow diagram illustrating the method for carrying out the union function 2406. Initially, several counters are set at step 2500. COUNT and BANK, the table access counter 2314, are both set equal to one (1) and SUM is set equal to zero (0). MAX is set equal to the largest integer count. At step 2502, the maximum value of the first COUNT and first BANK position between the first USE-COUNT TABLE and second USE-COUNT TABLE is determined and stored at RESULT. RESULT is added to SUM at step 2504. Then RESULT is stored in the corresponding location of table 2412. Next, COUNT is incremented by one (1) at step 2506, and inquiry step 2508 determines if the value of COUNT has exceeded the size of the largest bank position. If the value of COUNT is less than the size of the bank position control passes back to step 2502 to determine the maximum value of the next position within the bank of the USE-COUNT TABLES A 2402 and B 2404. The RESULT is stored in TABLE C 2412. Once the maximum value of all symbol positions within a bank have been determined by (COUNT>BANK size) at 2508, control passes to step 2510 where COUNT is set equal to one (1). If SUM is less than MAX the value of the MAX is set equal to SUM. SUM is then set equal to zero (0), and BANK is incremented by one (1). Inquiry step 2512 determines if another bank exists, and proceeds to step 2502 to determine the maximum values within the next bank. If other banks do not exist, the value of MAX is output at step 2514 into maximum set size 2415 (FIG. 23). The value of MAX stored in maximum set size 2415 (FIG. 23) represents the maximum number of RECORD MEMORY 78 entries that could result from a union of the key sets of the RECORD MEMORIES corresponding to USE-COUNT TABLES A and B. That is a union of the keys of RECORD MEMORIES A and B can result in a RECORD MEMORY C with no more than MAX key records.

Detailed Description Text (169):
COUNT is incremented at step 2706 and inquiry step 2708 determines if COUNT is greater than the size of the present bank. If COUNT is less than or equal to BANK SIZE then all symbol positions within a bank have not been examined and control returns to step 2702. If all symbol positions have been examined, control passes to step 2710, COUNT is reset to one (1) and BANK is incremented by one (1). Also at step 2710, if SUM is less than the value of MAX, MAX is set equal to SUM then SUM is set equal to zero (0). Inquiry step 2712 determines if another bank exists to be examined. If another bank exists control passes to step 2702. If no other banks are present, the value of MAX is output at step 2714 to indicate the maximum number of key records that can be in the key record memory resulting from the mask operation.

Detailed Description Text (170):
Referring now to FIG. 27, a series of USE-COUNT or INDEX tables 2800(a) through 2800(k) may be combined by a series of set operations into a result table 2802. This sequence of set operations is performed on like tables (either USE-COUNT or INDEX) using the associative set processor 2400. Initially, the RESULT table 2802 has all its entries set to the counts in the first table 2800(a). The table select switch 2810 selects the next table 2800(b) and the set operation is performed with set A stored in the result table 2802. Result table 2802 holds the USE-COUNT results of the operation between the first two record memories tables, for example the results of A intersect B. The input table selector switch 2810 is then set to the third USE-COUNT Table C 2800(c) and the set operation with the result table 2802 is preformed, for example R union C. is (A intersection B) union C. The input and output table selection switches (2810 and 2812) are controlled by the associative set processor 2400 and sequence through all the USE-COUNT tables for all the record memory to be combined according to a user specified sequence of operations to perform the required set operations on each input USE-COUNT table. The result of each set operation includes a MAX value 2806 which is the maximum number of key records which meet the combined set operations. If during the processing, no union operations are left to be performed and the MAX value is zero (0), then there are no records in the resulting set, all the values in one bank of the result table 2802 are zero (0), and processing may stop with a null result. If after processing all the input tables, MAX is greater than zero (0), then the result table 2802 stores the resulting USE-COUNT table. This resulting USE-COUNT table may be used to filter all the original record memories to produce the record memory data set resulting from the sequence of operations. The resulting record memory can have no more than the final MAX records.

CLAIMS:

1. A method for finding a record of data in a record addressable location of memory using valid data keys comprised of a plurality of values associated with the record, said method further pointing to the location of related data for invalid data keys, the method comprising the steps of:

assigning index values to valid and invalid symbol values within an entire key set prior to receiving a first key;

receiving a key associated with a record of data stored in a memory having record addresses;

arithmetically coding a key to a record index value;

if the key is valid, providing the record index value to the memory as an address, the record associated with the key being stored at the address;

if the key is invalid, providing a record index value pointing to a record stored in the memory which is related to a record associated with the key but not stored in the memory; and

accessing the record of data in the memory.

2. The method of claim 1, wherein the step of assigning index values includes the steps of:

dividing every key of a key set into a sequence of symbols;

counting each use of a symbol value within each symbol position of the key sets;

storing the count within a use count table;

assigning index values to each symbol value within a symbol position having a non zero use count; and

assigning index values to each symbol value within a symbol position having a zero use count equal to the index value of a non-zero use count symbol value most immediately following the zero use count symbol value.

3. The method of claim 1, wherein the step of assigning index values includes the steps of:

dividing every key of a key set into a sequence of symbols;

counting each use of a symbol value within each symbol position of the key set;

storing the count within a use count table;

counting the number of non zero entries within a symbol position of the use count table;

determining a base value for the symbol position; and

assigning to each symbol value with a non zero use count within a symbol position an index value equal to the count of the non zero entries from the beginning of the symbol position multiplied by the base value for the symbol position; and

assigning to each symbol value with a zero use count within a symbol position, an index value equal to the value of the non zero symbol position immediately following the zero use count symbol position.

4. The method of claim 1, wherein the step of assigning index values includes the steps of:

dividing every key of a key set into a sequence of symbol positions;

counting each use of a symbol value within each symbol position of the key sets;

storing the count within a use count table;

counting the number of non zero entries within a symbol position of the use count table;

determining a base value for each symbol position; and

assigning to each symbol value within a symbol position, an index value equal to the base value for the symbol position plus the count of the previous non zero entries from the beginning of the symbol position, multiplied by the base value for the symbol position.

5. An apparatus for performing set and relational operations on a plurality of sets of data records, the apparatus including:

a plurality of memory locations for storing a use count table for each of a plurality of sets of records, each record in each of the sets of records being associated with a key and the use count table for a particular set of records storing the number of times a symbol occurs in a particular position in any of the keys for the records in that particular set;

an associative processor for performing set and relational operations on the use count tables and, in response thereto, creating a use count table for keys of resultant set of data records which would result from operation on the plurality of

sets of data records.

8. The apparatus of claim 5, wherein the associative set processor includes a table access counter for sequencing through all index values resulting from a sequence of set and relational operations.

12. A method for determining a maximum number of key records resulting from the intersection of a first and a second key record memory, comprising the steps of:

determining the sum of use counts for each symbol position;

determining which symbol position has the smallest sum value of use counts; and

outputting the smallest sum value of use counts as the maximum number of key records resulting from the intersection of a first use count table and a second use count table.

15. A method for performing a mask operation between a mask table and a use count table comprising the steps of:

comparing the use count table to the mask table;

determining each non zero entry in the mask table;

storing in a corresponding position of a results table the use count table entry for each non zero entry of the mask table;

storing in a corresponding position of the results table a zero for each zero entry of the mask table;

determining the sum of the resulting use counts for each symbol position;

determining the minimum sum value; and

outputting the minimum sum value as the maximum number of key record entries resulting from the mask operation.

18. A method for encoding keys for use in an associative memory of a data processing system, each key including a plurality of symbols, each symbol having a predetermined symbol value and occupying a predefined symbol position; the method comprising the steps of:

encoding each key in a key set, the step of encoding a key in the key set including assigning and storing in an index table stored in a memory of a data processing system an index value for each symbol present in at least one symbol position of the key such that index values for all symbols in the key to which an index value is assigned compute to a record index value uniquely associated with that key; and

filling in index values in the index table for the at least one symbol position, the step of filling in index values including, for each symbol in a predetermined set of symbols which is not present in the at least one symbol position in any key in the key set, assigning and storing an index value equal to the index value assigned to a next adjacent symbol present in the at least one symbol position in any encoded key.

19. The method of claim 18 further comprising the steps of:

receiving an unencoded key;

returning a record index computed from the index values stored in the index table

for symbols present in the unencoded key.

21. The method of claim 18 further comprising the steps of:

for each symbol present in the at least one symbol position in any of the keys in the key set, setting a flag in the memory of the data processing system to a first value; and

for each symbol not present in the at least one of symbol positions in any key in the key set, setting a flag in memory of the data processing system to a second value.

22. The method of claim 21 further including the steps of:

receiving an unencoded key;

returning a record index computed from the index values stored in the index table for symbols present in the unencoded key; and

returning an indication of whether any of the symbols in the unencoded key for which an index value is assigned is a symbol not present in the key set.

23. The method of claim 18 further comprising:

receiving an unencoded key;

returning a record index computed from the index values stored in the index table for symbols present in the unencoded key;

returning an indication of whether any of the symbols in the unencoded key for which an index value is assigned is a symbol not present in the key set; and

storing data associated with the unencoded key in a first location in the memory identified by the record index if data associated with another key is not stored at the first location.

24. The method of claim 23 further comprising:

if there is data stored at the first location associated with a key other than the unencoded key, encoding the unencoded key by assigning and storing in the index table an index value for a symbol in the unencoded key not found in any previously encoded key such that the index values for all symbols in the unencoded key compute to a record index value uniquely associated with that key; and

storing the data associated with the unencoded key at a second location in the memory indicated by the unencoded key's record index value.

25. The method of claim 24 further comprising determining whether any one of the previously encoded keys has a changed record index value as a consequence of encoding the unencoded key and, if so, moving the record associated with the one of the previously encoded keys to a third location in the memory identified by the changed record index value.

28. A data processing system having an associative memory in which data is accessed using keys associated with the data, each key including a plurality of symbols, each symbol having a predetermined symbol value and occupying a predefined symbol position; the data processing system comprising:

means for encoding each key in a key set, the means for encoding a key in the key set including means for assigning and storing in an index table in a memory of the

data processing system an index value for each symbol present in at least one symbol position of the key such that index values for all symbols in the key to which an index value is assigned compute to a value uniquely associated with that key; and

means for filling in index values in the index table for the at least one symbol position, the means for filling in index values including means for assigning and storing for each symbol in a predetermined set of symbols which is not present in the at least one symbol position in any key in the key set an index value equal to the index value assigned to a next lowest order valid symbol.

29. The data processing system of claim 28 further comprising:

means for receiving an unencoded key; and

means for computing a record index from the index values stored in the index table for symbols present in the unencoded key.

30. The data processing system of claim 29 further comprising:

means for indicating whether any of the symbols in the unencoded key for which an index value is assigned is a symbol not present in the key set; and

means for storing data associated with the unencoded key in a first location in the memory identified by the record index if data associated with another key is not stored at the first location.

Previous Doc Next Doc Go to Doc#